VC++: A Visual Programming Language Framework

by

P. CHENNA REDDY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY 1997

VC++: A Visual Progrmming Language Framework

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

P. Chenna Reddy

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

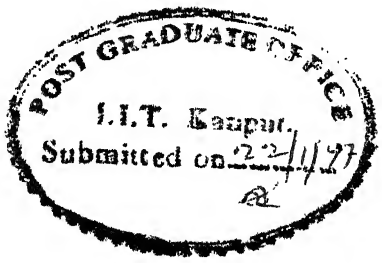
Jan. 1997

CERTIFICATE

This is to certify that the work contained in the thesis titled *VC++: A Visual Programming Language Framework* by P. Chenna Reddy has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

for Dr. Tapas K. Nayak,
Assistant Professor,
Dept. of CSE,
IIT Kanpur.

Dr. Ratan K. Ghosh,
Associate Professor,
Dept. of CSE,
IIT Kanpur.



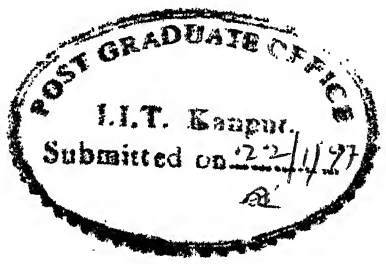
CSE-1997-M-RED-VIS

CERTIFICATE

This is to certify that the work contained in the thesis titled *VC++: A Visual Programming Language Framework* by P. Chenna Reddy has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Tapas K. Nayak,
Assistant Professor,
Dept. of CSE,
IIT Kanpur.

Dr. Ratan K. Ghosh,
Associate Professor,
Dept. of CSE,
IIT Kanpur.



Abstract

VC++ is a programming language framework based on the object paradigm. It offers one integrated programming model for both visualizations and general purpose computation. VC++ borrows and extends the C++ object definition facility called the class. More explicitly, the C++ class is augmented with a new section called "ecarules" for the specification of reactive behaviour of the objects. Facilities for creating parallel classes and concurrent objects have been included in VC++. The system was tested by building two nontrivial visual simulation environments namely, Eco-system Pond and City Traffic Simulation. The framework can be adopted with appropriate future extensions to serve as a development platform for Visual Programming Languages. In fact, with the current implementation of VC++ grade sheet problem [KASC95] has been programmed to indicate potentials of the proposed framework in building domain specific Visual Programming Languages.

Acknowledgments

I am extremely grateful to my thesis supervisors Dr. Tapas Kumar Nayak and Dr. Ratan Kumar Ghosh for their constant guidance throughout my work. My sincere thanks to Dr. T.V.Prabhakar, for his invaluable assistance during my stay at IIT(K). My sincere thanks to all my classmates who made my stay at IIT(K), a memorable one. My sincere thanks to Singuru, on whose work I could extend. My special thanks to djai, ssm, goel, kommu, lade. and gsri. Finally, my heartfelt thanks to my parents and my two brothers for their invaluable moral support and encouragement.

Contents

Chapter 1

Introduction

The use of graphics in programs and data dates back to 1950s when systems for automatic creation of flowcharts were built [Hai59]. There are many systems that either use program visualization or visual programming [MP85, cP86]. Program visualization refers to use of graphics for illustration of programs and/or data, whereas visual programming offers representative graphical syntax for ease in creation of programs. Therefore, visual programming paradigm narrows the gap in translating user's conceptual models to programs. In other words, visual languages lower the threshold of programming for end users. Therefore, both use and reprogramming of end-user tools become easy.

1.1 Motivation and Objectives

There is an apparent conflict in two extremities of the requirements, namely, the ease of use and the power of the language. The domain specific visual languages exhibit the ease of use in programming by visuals. Certain basic visuals are provided in the form of agents or programmable icons in such systems. These visual objects can be tuned to suit the end users requirements.

Another category visual languages is tailored to specific applications. For example, *Formal* [Shu85], is a VPL suited for data processing applications. The user

may draw the input and output forms specifying how values in input form influence the output form. Similarly, we have *Prograph* [MP85], *Fabrik* [ea88], *LabView* [KASC95], *Showandtell* [ea86], for a simple data flow or functional computation.

However, such systems are limited in power by the set of basic objects built into the language. Therefore, in order to address a different domain or a non-trivial extension of the same application domain to enlarger the scope, it becomes necessary to change or extend the set of primitives. Such modifications may be quite complex and may even call for a redesign of the existing visual system from the scratch. In other words, the programming power being sought can only be provided by a general purpose language. Unfortunately, it becomes too complex for a less trained user to write a program in a general purpose language like C++ or Visual C++. In fact, even for a minor change in the requirement the code has to be rewritten.

It is therefore, important as well as challenging to build a framework which may cater to both the above extremes. A visual programming environment [RS95] proposes to find such an in-between. In this work we present a visual programming environment VC++ based on C++. It provides a useful unified environment in which one may easily design new visual objects reusing existing visual objects. The relationships and the interactions of such visual objects can be specified using Event-Condition-Action (ECA) rules which are simple to specify and easy to modify. One can define concurrent as well as static visual objects which can interact among themselves. In our work, we have concentrated on Visual Programming, where the objects of concern have visual representations and their interactions have visual implications. Examples include CAD, City Traffic simulator, Aircraft simulator, etc. Different application domains require different visual objects with different representations as well as semantics. These reasons motivated us to design and implement an object oriented Visual Programming Environment, which can be used as a kernel to many domain oriented and end-user VPLs . We have tried in our work to identify and design a framework for such various domain specific visual programming tools . We have looked for a minimally adequate set of useful features in VC++ which would easily design such tools.

1.2 Related Work

In the past, there had been several approaches studied and implemented for visual programming. These approaches can be broadly classified into domain specific environments like HyperCard for UDUIs and PLAY system for children, or general programming environments like *Agentsheets, Prograph, Garden system* etc. The semantic models of these visual languages are either *data-flow, form based* or *generalized icons* based. A data-flow based visual program is a nested directed graph, in which nodes represent an operand or an operator and the arrows represent the control sequence. A form based visual program is an extension of a *spreadsheet*, in which a two dimensional cell layout is used while an icon based program consists of different spatial arrangement of icons with underlying meanings from an iconic system.

Visual C++ is Microsoft's C++ compiler/environment. It's got some good visual tools for drawing dialogs and such for Windows and a code generator that makes writing applications easier in some cases. Except for this, there is no difference between regular C++ and visual C++ per se. This is not a true visual language since the programmer has to still specify most of the semantic behaviour in a programming language (C/C++).

PLAY [kC86] is an interesting iconic visual programming system for children. It is an experimental computing environment for children that makes it possible to design characters, their movements and backgrounds and to make up plays and view animated performances of them. Each play is represented graphically by a script, which consists of a sequence of iconic sentences; the script appears somewhat like a comic strip.

The *GARDEN* System [Rei86] developed at Brown is a step towards a full scale visual programming environment designed for conceptual programming. It provides a framework supporting a wide variety of different graphical program views on a consistent manner. It provides an Object Oriented Programming environment with objects representing pictures that can be executed. It offers additional facilities that allow complex display of structures described by objects. It provides features that can be used to implement a new graphical language. This system shares the goals of

our VC++ system. However it lacks features that help creation and manipulation of concurrent objects and their inter communications.

Prograph programming language is based on the visualization of data-flow and functional programming. Its two dimensional visual syntax simplifies program construction. It combines Object Oriented Programming and data-flow programming paradigms for general purpose computations, including GUI applications. However, Prograph's language components use constructs such as iteration and therefore still require users to build up problem domain behaviours by assembling lower level primitives.

Agentsheets [RS95] is a Macintosh application implemented in Common Lisp. It is a system which supports the collaborative design of domain oriented visual programming languages in areas such as art, artificial life, education and environmental design. It provides role specific views and tools to meet the specific needs of designers. Visual languages created with Agentsheets consist of autonomous, communicating agents organized in a grid similar to a spreadsheet. Designers create visual programming languages by defining the look and behaviour of agents. Users interact with agents through direct manipulation.

Lab View is a visual programming application language. In LabView programs are called virtual instruments. In this programs are constructed using data-flow model. Programs are represented as data-flow graphs of interconnected icons. The LabView system is incorporated with graphic language to write programs.

Fabrik[ea88] is an interactive graphical programming environment designed to simplify the programming process by integrating the user interface, the programming language and its representation, and environmental languages used to construct and debug programs.

Pursuit is a new visual programming language and environment that serves as a form of feedback and representation in a Programming by Demonstration system. The language differs from existing visual languages because it explicitly represents data objects and implicitly represents operations by changes in data objects. The system was designed to provide non-programmers with programming support for common, repetitive tasks.

Phred is a visual parallel programming language in which programs can be statically analyzed for deterministic behaviour. The Phred programmer uses graph constructs for describing parallelism, synchronization and data sharing. Phred addresses the issues of determinacy by visually indicating regions of a program where nondeterminacy may exist. It also supports distribution of the programming environment over several workstations.

There are other similar visual programming languages. But many of these languages provide semantic constructs that are suitable to a particular domain or provide constructs that are not general enough to be applicable for any visual environment. In particular none of the above environments provide features for concurrency. The challenge however, is to come up with a mechanism for generalizing a set of features with general applicability. So in this work, we have tried to delineate features that are in general essential in the design of any visual language or environment.

1.3 Overview of Thesis

An earlier implementation of VC++ was done by Singuru [Kri96]. Unfortunately, the framework provided by his implementation had a number of limitations including serious performance drawbacks. The present implementation fixes the performance drawbacks and extends the framework with a number of new constructs. The notable features of current implementation and its major differences with respect to Singuru's version of VC++ are as follows.

- New constructs have been added for the specification of visual objects, namely *views*, *components*, and *owndisplay*. The construct *views* allows the user to specify different views or frames associated with an object. For example, in interactive environment smooth transitions of homogeneous visual objects are desired. To achieve this the user may define a view by explicitly specifying a *paint* function. The construct *components* allows the user to build complex objects from a number of different atomic visual objects. The tag *owndisplay* allows the user to define his/her specialized display function for the complex objects. By default VC++ system creates a display function.

Phred is a visual parallel programming language in which programs can be statically analyzed for deterministic behaviour. The Phred programmer uses graph constructs for describing parallelism, synchronization and data sharing. Phred addresses the issues of determinacy by visually indicating regions of a program where nondeterminacy may exist. It also supports distribution of the programming environment over several workstations.

There are other similar visual programming languages. But many of these languages provide semantic constructs that are suitable to a particular domain or provide constructs that are not general enough to be applicable for any visual environment. In particular none of the above environments provide features for concurrency. The challenge however, is to come up with a mechanism for generalizing a set of features with general applicability. So in this work, we have tried to delineate features that are in general essential in the design of any visual language or environment.

1.3 Overview of Thesis

An earlier implementation of VC++ was done by Singuru [Kri96]. Unfortunately, the framework provided by his implementation had a number of limitations including serious performance drawbacks. The present implementation fixes the performance drawbacks and extends the framework with a number of new constructs. The notable features of current implementation and its major differences with respect to Singuru's version of VC++ are as follows.

- New constructs have been added for the specification of visual objects, namely *views*, *components*, and *owndisplay*. The construct *views* allows the user to specify different views or frames associated with an object. For example, in interactive environment smooth transitions of homogeneous visual objects are desired. To achieve this the user may define a view by explicitly specifying a *paint* function. The construct *components* allows the user to build complex objects from a number of different atomic visual objects. The tag *owndisplay* allows the user to define his/her specialized display function for the complex objects. By default VC++ system creates a display function.

- A new message exchange facility have been added, namely, *async* with time out. This facility allows to specify any time dependent actions. If the action is not scheduled within the time bound then the action will not be executed. It speeds up visual interactions by eliminating some of the time dependent actions.

- We changed the semantics of *sync multicast* facility. Previously it allowed only group members to participate in exchange of messages. In the current implementation, the other visual objects not belonging to group are allowed to participate in exchanging information. But these objects would not normally wait for the results, unless wait for reply is specified.

- The visual objects have been endowed with a new dimension, i.e., the event dimension. The users can specify reactive behaviour to visual objects using ECA construct. We have designed some event composition operators for the composition of events. Four types of basic events have been considered namely, method execution events, external events (Mouse events), time dependent events, and user defined evens. This work contributes the major part of thesis.

- The system performance was improved by eliminating busy waiting state of the main thread for threads termination of concurrent objects. In the earlier implementation the user_main method was used to create threads for the concurrent objects in response to the external start button event. The user_main method is kept in busy wait for threads to terminate or the program to terminate. This approach had a serious drawback, the external events could not be monitored in time as the events are raised. In current implementation the user_main method is terminated immediately after creation of the threads for concurrent objects. The interaction among threads is accomplished by thread synchronization mechanisms. More explicitly, we use the system provided thread create function for creating threads then signal all threads to start execution. VC++ system creates detach_threads method to detach all the threads that were created by user_main method. This was used by the done

button event to terminate the threads. Once the done button event is raised the threads push the thread_detach function to their slow buffers. This way all the remaining methods pertaining a thread in its fast buffer or in its slow buffer prior to thread_detach function will be executed before the thread is finally terminated. This approach exhibits better overall performance compared to earlier implementation.

1.4 Thesis Organization

Chapter 2 describes the programming model. It also gives the design and implementation details of a visual environments named *City Traffic Simulation and Eco system Pond*, developed using this technique.

Chapter 3 discusses about the design and implementation details of the VC++ framework.

Chapter 4 describes about the parser developed for VC++.

Chapter 5 gives the design of two existing visual environments using the VC++ framework.

Chapter 6 concludes the work and suggests future extensions that can be made to the present system.

Appendix A gives the *main* part of the header files for each of the four basic classes, together with a detailed description of the interfaces they provide.

Appendix B gives the detailed header files for problems discussed through out the thesis report.

Chapter 2

VC++ Visual Programming Model

VC++ provides a visual programming model based on the notion of visual objects. A visual object is one which has a visual representation which may or may not change depending on the state of the object. Actions of a visual object may also be visible through its visual activities, such a visual object acts and interacts with other visual objects according to certain rules. Constructing a visual program involves construction of such visual objects and specification of their visual actions and interactions.

In order to illustrate the concepts of VC++ we mainly discuss two examples throughout the paper. These two examples are City Traffic Simulation and Ecosystem Pond described as follows.

In City Traffic Simulation, objects representing cars(vehicle) move forward when a stretch of road is in front of them and obey the traffic rules. The other objects in City Traffic Simulation are traffic lights that keep on changing states (red, green, amber) at a predefined rate. The detailed behaviour of the objects (i.e., actions and interactions with other objects) are given below.

The behaviour of the vehicle is :

- After the vehicle is started, check whether it is about to collide with any other vehicle and also check whether the traffic light is green or not when it is close

to the traffic light.

- If the vehicle is about to collide with any other vehicle then don't move the vehicle.
- If the relevant traffic light is red then join the vehicle in the corresponding waiting group.
- If the vehicle is not likely to collide with any other vehicle and the relevant light is green then move the vehicle.
- If the vehicle reaches a four road junction take a random decision to go either in left, right or straight directions.

The behaviour of the Traffic Light is:

- Keep on changing the state of the Traffic light at some predefined rate.
- As the light turns green start all vehicles which are waiting for the light to turn green.

In Ecosystem Pond, objects representing big fishes and small fishes move forward according to the following rule :

- If a small fish comes into the vicinity of a big fish then it runs away from the big fish(that means it changes its direction).
- Big fish eats small fish if the big fish sees the small fish in its vicinity.
- Objects representing water-bubbles are generated randomly and move upwards. As they move up they grow in size.

From the user perspective, visual programs/environments developed by any VPL consists of visual objects. In City Traffic simulation, the visual objects are cars, traffic lights, roads, on the other hand the visual objects are fishes, and water bubbles in Ecosystem Pond.

VC++ provides one integrated model for both visualization and general purpose computation. We use the C++ object model, as the basis for the object model of VC++ [ES90]. In VC++ all the classes of visual objects are derived instances of *VisualObjClass*. VC++ extends C++ classes by providing facilities for creating and manipulating concurrent objects and associating event-condition-actions (E-C-A)with objects. VC++ provides facilities for creating complex visual objects. The constructs of VC++ that support the above facilities are discussed in the following sections.

2.1 Visual Objects

Object modelling in an Object Oriented system usually refers to three aspects, namely

- Structural Modelling
- Behavioural Modelling
- Event Modelling

Structural modelling refers to how to construct the object, behavioural modelling refers to how to serve different message requests from other objects and event modelling refers to the reactive behaviour when events of interest happen. Visual object modelling extends all the above three aspects in visual dimension. The various constructs of VC++ supporting these aspects in visual dimensions are discussed below.

2.1.1 Structural Modelling

A visual object must have a visual representation which may either be an icon or a drawing. VC++ allows the user to specify his/her own visualizations. A complex object may be made up of components. A component is another visual object which constitutes a part of the complex visual object. The relationship between object and components may be dynamic i.e. at run time some components may join an object

or may leave an object. The geometry of components are defined with reference to the geometry of the containing object. All the visual objects are associated to a single window, called the presentation window.

Visual object may have different views at different times. The relationship between object and view may be dynamic i.e. at run time the view of an object may change or different views may be added to the object. The notion of views may be used, for example, as frames for animation. In order to have good animation we need to have different views/frames for the objects, eg. in case of Ecosystem pond we use two frames as views to have motion of the fishes.

VC++ provides tags for the specification of object/component and object/view relationship. The tags are *components* and *views*. For the dynamic relationship of components, it provides two methods namely *push_component*, and *remove_component*. The *push_component* method allows the user to push new component into the complex object, and *remove_component* allows the user to remove the component from the complex object.

Display of views is taken care of by VC++. VC++ generates a display function using the *paint()* functions of different views involved. Every visual object class should define a function *paint()*. However, the designer of the class may write his/her own display management. The class qualifier *owndisplay* forbids VC++ to generate *display()* routine. If the class definition doesn't precede with tag *owndisplay*, VC++ creates the display function for the complex object. For the dynamic relationship of objects and views, it provides five functions namely *set_current_view*, *get_current_view*, *add_view*, *no_of_views*, and *delete_view*. The *set_current_view* function allows the user to change the current view. Views are indexed, and may be referred to by the indexes. If the views are specified statically during class definition they are assigned indexes as 0, 1, 2 in the order of specification. The *get_current_view* returns the current *view_id*. The *add_view* adds new view to the object and returns the *view_id*. The *delete_view* deletes a view from the object *view_list*. The function *no_of_views* returns the number of views associated to an object.

Example 1: Consider the City Traffic Simulation. The visual object road_map is made up of different visual objects namely `road1`, `road2`, and `road3`; where

road1, road2, and road3 are instances of the class Road. The class declaration of this visual object in VC++ is as given below.

```
class Road;
class Roadmap: public VisualObjClass
{
   private:
              ⋮
      components  Road road1(10,100), road2(100,150), road3(150,100);
              ⋮
};
```

A roadmap thus consists of three roads: road1, road2, road3. The road1 is created at the point (10,100) with reference to the origin of the roadmap instance. Similarly road2 is created at (100, 150) and road3 at (150, 100).

In the above example, VC++ automatically creates the display function for this class definition. If the user is interested to have his/her own display function the class definition must prefixed by *owndisplay.*

If the roadmap1 is an instance of the Roadmap class, one may add another instance road4 (pointer type) to roadmap1, as

```
    roadmap1->push_component(road4);
```

or

```
    roadmap1.push_component(road4);
```

depending on whether roadmap1 is a pointer type or not.

Similarly for removing a component, remove_component would be used in place of push_component.

Example 2: Consider the Ecosystem Pond. The visual object fish has two frames. Frame1 and Frame2 are displayed alternatively in course of motion of a fish. In VC++ the two frames are modelled as two views as follows.

```
class sea_creatures : public VisualObjClass
{
    private:
        int x, y; // myoffsets

            ⋮

        views sea_creatures(1,x,y,2), /* frame type 2 */
              sea_creatures(1,x,y+100,1);/* frame type 1 */

            ⋮

     public:
        sea_creatures(int fish_no, int x_offset, int y_offset,
                      int frame_no);

            ⋮

};
```

In the above example `fish_no` indicates the type of the fish i.e. either big fish or small fish; `x_offset` and `y_offset` specify the position with respect to window; `frame_no` specifies the frame number. In this example we have defined statically the views/frames of the object. The user has the flexibility to add new views to the object or delete views. If `fish1` was an instance of the `sea_creatures` class, to add another view `fish_ver`(pointer type) to `fish1`, the specification would be as given below.

```
    v_id = fish1->add_view(fish_ver);
or
    v_id = fish1.add_view(fish_ver);
```

depending on whether `fish1` is a pointer type or not.

Similarly a view may be deleted using method *delete_view*. The *delete_view* method requires the index of views to be deleted.

2.1.2 Behavioural Modelling

As in C++ the behaviour of an object can be specified using methods or messages. By execution of a method, for example move, the display representation of the object may be affected. VC++ controls visual effect of behaviours using the qualifier *param* for variables. If a method changes a *param* variable it raises an appropriate event and the display representation of that object changes.

Example 3: Consider the City Traffic Simulation. The cars move forward along the road by using the method `move()`. The following specification shows the association of the method move with the display parameters (*param*)

```
parallel_class Vehicle : public VisualObjClass
{
  private:
              ⋮
    param  int car_pos;
              ⋮
  public:
    void move(); // method to move
              ⋮
};

void Vehicle :: move()
{


   // x and y represents the position of the object
   // on the display
          x = x + 10;
          car_pos = car_pos + 10;
```

```
    ⋮
}
```

The display parameter car_pos is a variable on which the presentation of a car depends. The method move affects this variable, thereby affecting the presentation of the car.

The method move changes the position of the object on the screen. VC++ checks whether the position change is sufficient enough or not in order to have smooth transitions and avoid unnecessary redrawing.

The display behaviour of a visual object is modelled by a vector of its *param* variables. If $x_1, \ldots x_n$ are the param variables then the vector X = $[x_1, \ldots, x_n]$ is called the display behaviour vector of a visual object. As x_i changes by Δx_i the object may have to be redisplayed. However, if the object is redisplayed even for very small changes in param variables then it may not be desirable because the effect on the display may be imperceptible. Therefore, an object is redisplayed only when its display behaviour vector changes at least by some threshold value. The change $|\Delta X|$ is computed as a function of Δx_i, $1 \leq i \leq n$. Thus $|\Delta X| = f([\Delta x_1 \ldots x_n])$, where f is a function called the *threshold function* available with the respective visual class. A threshold parameter τ is assigned to each object. Threshold τ is assigned at the time of creation of the object, using its constructor. On a change of some param variable, an object may be redisplayed only when $|\Delta X| > \tau$.

However, this scheme may require unnecessarily many invocations of threshold function. To reduce this overhead a vector of param threshold values(ptv) is provided as α_i for x_i. If $\Delta x_i < \alpha_i$ then the threshold function is not invoked and the object is not redisplayed.

2.1.3 Event Modelling

Event modelling is important in a visual environment. In many visual applications it is important to monitor situations of interest and to activate timely responses when the situations occur. The importance of event modelling and its representation is explained below with examples.

Example 4: Consider the situation in City Traffic Simulation when the traffic light, say the eastern one, changes its color from red to green. It has to wake up all the vehicles which are waiting at this traffic light. Thus the class `Light` (the class of traffic light objects) defines a rule named `e_event` which states that after the event of `change_lights` takes place all the objects waiting in a group `east_wait`, for moving eastwards should start. In the `Light` object variable `current` indicates the direction of the green light. The following rule illustrates this logic.

```
eca Light :: e_event
{
precond:
        after change_lights
postto:
        east_wait;
cond:
        current == EAST
action:
        async start();
}
```

Example 5: In City Traffic Simulation, the car that is about to move, checks whether it will collide with any other object or not. So in this case the system will raise an event before the car moves (i.e., after start) and it posts the appropriate method to all the cars in a group, named `car_grp`. The following rule illustrates this logic.

```
eca  Vehicle :: e_collide
{
   precond:
       after start
   post_to:
       car_grp
```

```
    cond:
    action:
        sync check_collide(object_info);
}
```

The check_collide is a method in the Vehicle class to check_whether it is colliding with the object, i.e. object whose information is passed to this method.
The event specification and event composition operators are discussed in section 2.4.

2.2 Parallel Class

VC++ visualizes an object as either static or concurrent. Static objects will not change their states concurrently with other objects. Concurrent objects change their states concurrently with other objects. This can be visualized by looking into the City Traffic Simulation problem. In that the object road-map is a static object and the cars are concurrent objects. Static objects are specified in a similar fashion as C++ objects. For the specification of concurrent object classes VC++ has a new notion of class definition called *parallel_class*. The objects of *parallel_class* may or may not be concurrent. In order to have concurrent objects, the object declaration must be prefixed with *concurrent*. The following example illustrates the constructs *parallel_class* and *concurrent*.
Example 6: Let us consider the class definition of Vehicle in City Traffic Simulation.

```
parallel_class Vehicle : public VisualObjClass
{
            :
};
```

Concurrent objects of class Vehicle can be specified as

```
concurrent Vehicle car1,car2;
```

or

```
concurrent Vehicle *car1; // car1 is a pointer
                          // to a concurrent Vehicle
```

A new dynamic instance may be created as

```
car1 = new concurrent Vehicle();
```

However, a static Vehicle object may be created as

```
Vehicle dead_car;
```

where the dead_car is a static object. It will not be moved concurrently with concurrent objects of the Vehicle class, and it will be not participate in the dynamic behaviour consisting of events and concurrent objects.

2.3 Message Facilities

VC++ provides message facilities in order to communicate among the objects. Depending upon the message timing i.e. urgency or non_urgency and number of copies sent i.e. multicast or simple, there are broadly four categories of messages. VC++ provides the following four types of message facilities.

- Asynchronous
  - the message is sent to one object and the sender doesn't wait for the result of the message. The execution of this message may depend on a time bound. The time bound is optional. The action must be scheduled before the time bound. If the action is scheduled after the time bound in that case the action will not be performed.
- Synchronous
  - the message is sent to one object and the sender waits for the result of the message.

- Asynchronous multicast
  - the message is sent to a group of objects and the sender doesn't wait for the results.
- Synchronous multicast
  - the message is sent to a group of objects and the sender waits for the results from all the objects.

Consider the City Traffic Simulation, whenever we want a car to start, a start message is sent to the car object. Then without bothering for the result of the message, some other car may be asked to start. This type of messaging is thus asynchronous. Now, the reactive behaviour of start method makes it check the chances of a collision with all the cars in case the move is carried out. For this purpose we require synchronous multicast message facility. On the other hand, in case Traffic light changes from red to green, the cars waiting for this signal will be conveyed start messages in asynchronous multicast mode by Traffic light.

For static objects the simple and multicast messages are always treated as synchronous.

In order to use the multicast facilities of VC++ we need to define a *group* to maintain the set of objects which constitute a communication group. A message may be multicast to a group by sending the message to every object in the group. Objects may join or leave a group. The methods for joining and leaving a group are *join_group(object_name), leave_group(object_name)* respectively.

Example 7: This example illustrates simple message facilities

```
concurrent Vehicle car1;
```

A move message is sent in async mode to `car1` as

```
async     car1.move();
```

A move message is sent in sync mode to `car1` as

```
car1.move();
```

Example 8: This example illustrates the use of multicast message facility of VC++.

```
Group <Vehicle> car_grp;
```

The above declaration creates new Group object `car_grp`.

In order to use the multicast messages the objects must join the group. Suppose we want to multicast the message `check_collide` to `car1`, `car2`, and `car3`. All these three cars must join a group. The cars may join the group `car_grp` as follows:

```
car_grp.join_group(car1);
car_grp.join_group(car2);
car_grp.join_group(car3);
```

The message `check_collide` is multicast in async mode to the group as

```
async car_grp.check_collide(object_info);
```

The same message may be multicast in synchronous mode as

```
car_grp.check_collide(object_info);
```

2.4 Events

Event modelling is very important in a visual environment. In a visual Object model, the objects may have independent existence. Objects may interact among themselves either asynchronously or synchronously. Some of the events are system(VC++) generated events and user can also define his/her own events.

Events are modelled using Event-Condition-Action(ECA) rules. The semantics of ECA rules is as follows : when the event occurs or is raised, evaluate the condition and if the condition is satisfied execute the action.

In ECA model, there are four types of coupling possible between each of E-C and C-A. These are immediate, deferred, separate dependent, separate independent[DBB+88, MD89]. In our implementation the E-C coupling is immediate, C-A coupling may be either deferred or immediate.

- **deferred:** After the condition is satisfied the execution of action posts to the appropriate object in *async* mode. There may or may not be a time bound for scheduling. If the time bound is not specified, in that case the action will be executed surely. If the time bound elapses before the action starts executing then in that case it will never be performed.

- **immediate:** After the condition is satisfied the execution of action posts to the appropriate object in *sync* mode.

Each event rule is defined by the following components:

- ***Precond:*** (Enabling condition)

  - defines when to consider the event.

    (This is specified to reduce the cost of the system rather than raising the events after the execution of all the methods on any object).

- ***Condition:***

  - this is the actual condition of the ECA rule. If the condition is true then the action is posted to the target objects or group. This condition may test object variables.

- ***Postto:*** (Target Object/Objects for the event)

  - to which object the action is to be posted. (the object may be itself or other object or object list or group).

- ***Action:***

  - this is the action which is to be executed if the event is raised and the condition is met.

  - the action may be synchronous or asynchronous. The qualifier for synchronous is *sync*. (This means the C-A coupling is immediate). The qualifier for asynchronous is *async*. (This means the C-A coupling is deferred).

The syntax of an ECA rule in BNF description is given as follows

eca classname:: event_id

{

precond: [[*after*] method_name]*

postto: [object list | group list | *self*]+

cond: composite_event | composite_logical_event

action:

[*sync* | *async* < *hrs* : *mins* : *secs* >] action_name

}

2.5 Event Structure

An event may be a basic event or a composite event.

2.5.1 Basic Events

There are four kinds of basic events: member function call events, time events, user defined events and external events(eg. Mouse events). Basic events are mutually disjoint. Member function call events, external events, and time events are posted automatically by the system. User defined events must be explicitly posted by an application.

- *Member function events*

Invoking public member functions on a visual object causes events to be posted to the object

For example, the call *p->f(...)*

where p is a visual object pointer and f is a public member function causes the event *after f*

to be posted to the object after the call.

Time Events

Some objects may be interested to execute some of the methods after some time or at specific point of time (i.e., the time may be relative or absolute). ECA allows the user to specify time events.

Time events in VC++ are

- *at* time_specification
  - refers to absolute time
  - this event will be raised exactly at the time specified.
    Example 9: at 5:0:0
    this event will be raised exactly at 5 hours of the day.
- *after* time_period
  - refers to relative time
  - this event will be raised exactly after the time specified.
    Example 10: after 0:0:5
    this event will be raised exactly after 5 secs from the start of the application.
- *every* time_period
  - refers to relative time
  - this event will be raised every time period from the start of the application.
    Example 11: every 0:0:5
    this event will be raised every 5 secs from the start of the application.

The time can be specified in the following format

time(HR= hours, MIN= minutes, SEC= seconds)

User-Defined events

Users can define new events, such events must be posted explicitly. For example,
event start;
defines a new event (not associated with any method).
User-defined events are posted to the visual object by calling the function PostEvent.
For example
PostEvent(p,start);
posts the event *start* to the visual object referenced by the pointer p.

External Events

Users can click mouse buttons on any of the objects on the display. A mouse button raises the mouse button event to the object that the user clicked upon.

Mouse button events in VC++ are:

- Mouse_Button_1_down
- Mouse_Button_1_up
- Mouse_Button_2_down
- Mouse_Button_2_up
- Mouse_Button_3_down
- Mouse_Button_3_up
- Mouse_Move

Example 12: The following ECA rule says that a vehicle object may disappear when mouse button1 is pressed on it.

```
eca Vehicle :: e_remove
{
precond:
postto:
```

```
  self;
cond:

          Mouse_Button_1_down
action:

          sync erase_display();
}
```

2.5.2 Composite Events

An event history, which is associated with every object, is a sequence of basic events that were posted to the object. Because of the assumption that the basic events are disjoint, the sequence is well defined. One or more basic events may be composed to form a composite event. A composite event is associated with the event history for its evaluation. Composite events are composed from basic events using event composition operators. Composite constructs used in VC++ are similar to the event specification in active databases[GJS92].

The event composition operators are

- *Sequence Operator*

The sequence operator is denoted by comma. Parenthesis must also be used to delimit the sequence expression. For example, event: (A,B)
is satisfied if the first basic event posted after A is B. A and B must be basic events.

- *Disjunction Operator*

Operator || is used to specify the occurrence of one of two events. For example, event: (A | | B)
is satisfied if either of the events happens.

Conjunction Operator

Event: $A\&\&B$
is satisfied if the events A and B are satisfied simultaneously by the posting of the same basic event. If A and B are both distinct events, then the composite event will never be satisfied. This follows from the disjointness of basic events.

Negation

Event: $\sim$ A
is satisfied by any basic event other than A.
Expression: $\sim (A||B)$
is similarly satisfied by any basic event other than A, B.

Repetition Operators

- The + operator:
  Event: +A
  is satisfied by a sequence of one or more occurrences of A events.

- *+any* operator:
  is satisfied by one or more occurrences of any event. For example, a b, a a, or a b c all these satisfies +any, where a ,b and c are basic events.

Count Repetition Operators

There are also three count repetition operators: *relative, every*, and *choose.*

1. **The *relative* operator :**
   Event: *relative*(n, A)
   specifies a composite event which is satisfied by the n th and succeeding occurrences of the basic event A.
   Example 13:
   Event: *relative*(3, A)

specifies a composite event which is satisfied by the third and succeeding occurrences of the basic event A.

2. **The *every* operator :**
   Event: *every*(n, A)
   is satisfied by the n th, the 2*n th, etc. occurrence of basic event A.
   Example 14:
   Event: *every*(3, A)
   is satisfied by the third, the sixth,etc. occurrence of basic event A.

3. **The *choose* operator :**
   Event: *choose*(n, A)
   is satisfied just after the n th occurrence of event A.
   Example 15:
   Event: *choose*(3, A)
   is satisfied just after the third occurrence of event A.

Time Composition Operators

There are three Time Composition Operators namely *then*, *then every*, and *between*.

- The *then* Operator:
  Event: (A then time_period)
  is satisfied after the *time_period* from the time when event A occurred.
  Example 16: Event: (A then 0:0:5)
  is satisfied after 5 seconds from the event occurrence of event A.

- The *then every* Operator:
  Event: (A then every time_period)
  is satisfied after every *time_period* from the occurrence of event A.
  Example 17: Event: (A then every 0:0:5)
  is satisfied after every 5 seconds from the occurrence of event A.

- The *between* Operator:
  Event: *between*(A,B)

is satisfied once between every occurrence of event A followed by event B.(not immediately)

Example 18: Event: *between*(A, B then 0:0:5)
is satisfied between the occurrence of event A followed by event B and a maximum period of 5 seconds thereafter.

Event Ordering Operators

There are also two event ordering operators: *relative* and *prior*.

1. **The *relative* operator :**
   Event: ***relative***(**A, B**)
   is satisfied for every occurrence of a B event occurred after the event A has occurred.(events A and B cannot overlap in time)

2. **The *prior* operator :**
   Event: ***prior***(**A, B**)
   is satisfied for every occurrence of a A event occurred before B event.(events A and B can overlap in time)

Chapter 3

VC++ Architecture

VC++ is designed and implemented using OOP techniques in which the whole system is structured as a set of interacting objects. There are five principal modules in VC++ architecture.

- VisualObjClass: responsible for defining visual objects.
- WindowClass: responsible for managing window containing visual objects.
- ConcObjClass: responsible for managing concurrent objects.
- Group Communication Module: responsible for managing communication among objects and groups.
- Event Manager: responsible for events.

Architecture of the VC++ system is shown in figure 1. The five different modules constitute the VC++ framework kernel. VC++ parser translates the VC++ language constructs to interpret them using the VC++ kernel. An application program is written in VC++ using VC++ language support.

In the VC++ kernel the arrows indicate the interactions among the modules. Window class manages windows and the visual objects placed in it. Concurrent objects receive/send messages through ConcObjClass and Group Communication

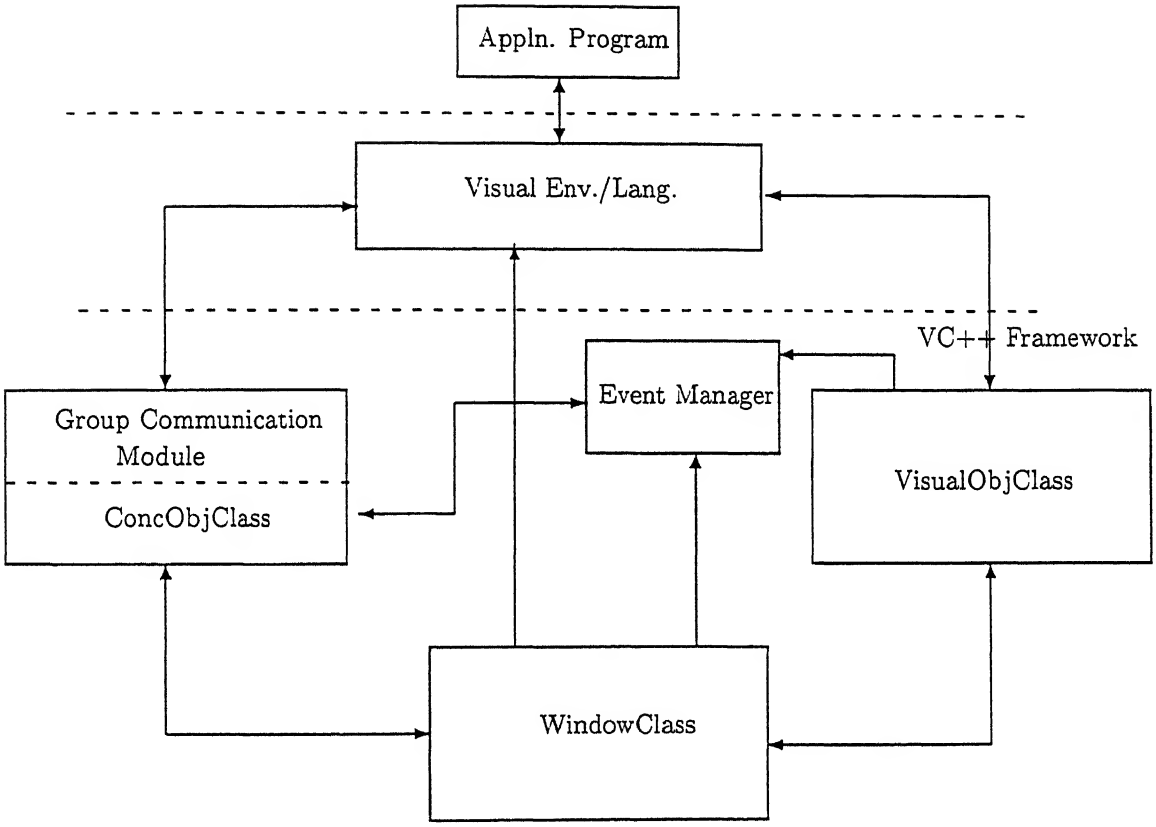


Figure 1: Architecture of the VC++ System

module. Execution of methods and display of objects is taken care of by VisualObjClass. Events may be raised by objects or the window events are taken care of by the event manager.

The implementation details and the interface provided by the five basic modules which form the kernel of VC++ are discussed in the following sections.

3.1 VisualObjClass

All user objects in VC++ are derived instances of this base class. It provides the minimal basic features needed by visual objects. Many of these features are implemented as virtual functions, so that user can redefine visual objects to suit their needs or use the primitive facilities already provided, if they are adequate.

The basic features provided by this class can be divided into three categories. They are

- Visualization Features
- Display Features
- Animation Features

3.1.1 Visualization Features

The objects in VC++ are logical objects with visual representations. Every object that is to be represented, occupies a certain area of the screen in which it is to be visualized (displayed). A bounding box or a rectangle can be used to constrain the area within which an object lies. As all objects in VC++ are either direct or derived instances of VisualObjClass. All such objects will have a bounding box. A bounding box is represented by a point, length and width. Operations required on a bounding box like initializing an object's bounding box with the given values, testing whether a given point lies within an object's area etc., are supported.

A complex object can be constructed from simpler component objects. This class provides the primitive features needed to construct a complex object from

its components. Further in certain simulation experiments, the components with which an object is constructed may change during run time[section 2.1.1]. The operations required to change the components list of a particular object dynamically are provided by this class.

3.1.2 Display Features

Each visual object has its own pictorial representation that depends on its structure. Some object may be displayed using bitmap images while others may be displayed using line drawing constructs. But every object have a method to display the particular object. So this class provides a *pure virtual display function* that must be defined to suit each object's display structure. If the object is a complex object, VC++ provides a default display function if the user has not interested to write his/her own display function. VC++ display function displays a visual object's view within its bounding box and also displays the component objects according to the component geometry. The VC++ display function for complex objects may be overridden in a class by prefixing the class definition with *owndisplay*. Every object in the program may not be visible at every instance of the execution of the program. So this class also includes a minimum form of erase (simply display the object in background color) that can be used to make an object invisible. Visual objects with complex display representations may require a more efficient form of erase. As this feature is provided as a virtual method, it can be redefined in the object's class, to suit its need. This class also provides features that can be used to allocate colors and images that minimize the amount of work required to produce a pictorial(display) representation of the object.VC++ provides three methods for color allocation and making image namely *FindColor, FindColorByName, and MakeImage*.

The display representation of an object may change due to changes in attributes of an object by the execution of an object's method. Such attributes may be represented using the *display param* variables of VC++ as discussed in section 2.1.2. The changes in display parameters raise appropriate events which are handled by this class to reflect the behaviour visually. The virtual *threshold* function provided in this class, checks whether the attribute changes are sufficiently high for object's

redisplay or not according to the model discussed in section 2.1.2. This aides smooth transition in object's display, by reducing flicker.

3.1.3 Animation Features

All dynamic environments require some form of animation [NT85] for its visual objects. Animation can be produced using a number of techniques. This class provides three forms of techniques for producing animation. They are Position Change, Shape Change, and Color Change.

Position Change: This class provides a primitive form of *move* function that simply moves the object by erasing its old position and displaying it in its new position. It uses the *threshold* function to check whether the position change is sufficiently high or not for redisplay. As this feature is provided by a virtual method, user visual classes may redefine it to produce more complex forms of motion control.

Shape Change: Animation can be produced by changing the position and size parameters of an object. By making such parameters as *display param* variables of VC++, implicit animation can be produced. This class as described earlier provides the required operations needed for *display param* variables. Also an object can be animated by showing its different views at different states or configurations. This can be done by associating alternative views of an object. VisualObjClass provides the features to specify the components of a main object and also operations that can be used to vary the components of an object dynamically at run time. It also provides the operations required to *zoomin or zoomout* a particular object.

Color Change: Objects can also be animated by changing dynamically certain colors, with which the original object is represented. This class provides the basic operations required to change color [JR94] representations of an object.

The header file for this class along with the detailed descriptions of interface is given in Appendix A.1.

3.2 WindowClass

WindowClass is designed in order to provide computer supported visual cooperative environment to the users. WindowClass acts as a house keeping agent for all visual objects. It maintains a list of all objects that are currently active in the users VC++ program window. It uses a *list* to maintain these objects. It communicates with the VisualObjClass through proper method invocation of the object instances of that class. Contrary to the VisualObjClass in which most of the methods are provided for the user of VC++, the methods in this class are used by the VC++ language itself rather than the user. In the present implementation of VC++, we are using a global object instance of this class that is initialized at startup for maintaining objects. This WindowClass instance acts as the service agent for all the visual objects created in it. The arrival and departure of visual objects is informed to this global object through its method invocations which are inserted at the appropriate places by the VC++ parser during preprocessing.

The basic features provided in this class are of two types, namely

- Maintenance Features
- Service Features.

3.2.1 Maintenance Features

This class keeps track of all the user objects declared and active within the user's program. We use the method *Place_in_window*, to place newly created visual object in the window. Once an object is created it occupies a place in the display window, although it may be invisible at certain instances of execution of the program. Similarly we use the method *Remove_from_window*, to delete an object from the window. This class provides the basic routines required to inform about an object's creation or deletion. These routines take care of displaying a newly created object in the window (i.e. placing in the window) and erasing of a deleted object (i.e. removing from the window).

3.2.2 Service Features

Window class acts as a guarding agent as it keeps track of global information. It should provide features that concern the global view of all the objects as a whole. It provides features to display/hide the active objects. It also provides routines such that at a given point, it can retrieve an object or show/remove the details of an object whose bounding box contains the given point. In this, if two or more object's bounding boxes overlap, it operates on the first such object in the list of objects it maintains. Objects are maintained in a last in first list.

WindowClass also acts as an external event generator. The window agent receives all the mouse events and determines the object_id whose bounding box contains the mouse position and it posts the appropriate event (eg. mouse_button_1_down) to that object. The header file for this class along with the detailed descriptions of interface is given in Appendix A.2.

3.3 ConcObjClass

Parallelism allows multiple activities to occur and interact simultaneously. Concurrency [BW95] is necessary to allow multiple visual object activities to occur at the same time. A concurrent object is executed by a thread created using DEC pthreads library [SMLM91, Dec]. All operations on a particular concurrent object are done by the thread that is alloted for it. A separate object instance of this ConcObjClass is alloted for each concurrent user object. Shared global memory is used as the media for communication.

This class deals with multiple threads of control and their synchronization. It also provides the basic features required in the communication of various concurrent objects.

3.3.1 Message Execution and Buffer Management

ConcObjClass manages dispatch and execution of messages by scheduling message processing jobs to worker threads. Each concurrent object is allocated a buffer called

job queue. Jobs(messages) for an object join its job queue.

A *worker* routine which works on behalf of each concurrent object, reads each job(a message) from the corresponding object's buffers in sequence and processes them. When there are no jobs, the worker simply waits until a job arrives. Th job queue synchronization problem is an instance of a *multiple writers and single reader* problem i.e. threads of the objects that want to send messages to a particular object (say P), write the messages in P's buffers and P's worker thread executes them one by one.

Two types of buffers namely an *urgent buffer* and a *normal buffer* are provided by this ConcObjClass. All *async* messages are written in the normal buffer, while all *sync* and *sync. multicast* messages, and and other messages that need a *return* value are written in the urgent buffer.

The *algorithm* for the worker routine which executes in a concurrent object thread schedules jobs from these two buffers in a priority scheme as follows.

< worker's algorithm >

Repeat

1. Execute all the jobs from the urgent buffer, one after the other.

2. Execute a single job from the normal buffer, if there is any.
   continue.

As can be seen in the algorithm for worker, the worker processing the jobs gives higher priority to jobs in the urgent buffer i.e. it executes jobs from normal buffer only if there are no pending jobs in the urgent buffer.

The synchronization routines needed to synchronize the multiple writers and the single reader problem in both the types of buffers is provided by the ConcObj-Class. These routines use the *mutexes* and *condition variables* to achieve the desired synchronization.

When an object wants to send a particular type of message to another object, it tries to write that message in the appropriate job buffer of the destination object. But when there is no room in the destination object's shared memory, the sender object has to wait until room is available. This class provides routines that allow

the receiver object to inform all waiting senders, whenever some room is available. Also whenever an object has no jobs to do, it simply waits. This class also provides features to inform the receiver, whenever a new job is sent to it.

Further one object may want to wait until another object finishes a particular job. This is the case when an object sends sync message or waits for the result of some message. This class provides the appropriate routines to achieve the desired synchronization. But an object idly waiting for another object to finish its job may get involved in a deadlock - eg. two objects may simultaneously wait on each other and produce a deadlock. So the worker routine provided in this class, rather than idly waiting, continue processing the remaining jobs, if any, for the waiting object.

In addition to these facilities, this class provides routines that allow an object to do its jobs by reading from the two different types of buffers one after the other in sequence. It also provides a routine that causes normal termination of a concurrent object and the thread executing it. This routine is scheduled as a normal job by VC++ after all jobs on this object are scheduled.

The header file for this class along with the detailed descriptions of interface is given in Appendix A.3.

3.4 Group Communication Module

As described above, the ConcObjClass provides the basic communication media as *shared global memory*, required for communication among various concurrent objects. All end to end VC++ communication statements like synchronous and asynchronous messaging between any two concurrent objects are handled by the ConcObjClass. The group communication module provides more complex form of communication facilities like asynchronous multicast and synchronous multicast. This module provides routines for forming a group, joining a group, leaving a group, checking a group strength and multicasting messages. A group Group $< T >$ is implemented as a template class. A specific group is created as an instance of the template class. The features provided by this class can be grouped into two categories. They are

- Group Formation Routines
- Multicast Routines

3.4.1 Group Formation Routines

Group formation routines allows an object to join or leave a group dynamically at run time. This class has internal routines to maintain the consistency for the groups. For eg. an object may not be allowed to join a group if the group is already full or an object may not be allowed to leave a group if it is not a current member.

3.4.2 Multicast Routines

Multicasting of a message through a group involves sending of the message to each and every current member of the group. In case of asynchronous multicast, message is sent to the members of the group and then the multicast is over. In case of synchronous multicast, it sends the message to the members of the group and waits until the messages are processed by all the receivers.

In both the forms of multicast, the protocol used by this class ensures that the multicast is an all source ordered multicast i.e. if two messages *m1* and *m2* are simultaneously multicast to a group G, then these messages will reach all the members in the same order, m2 after m1 or m1 after m2. This is achieved using a single buffer which is locked by the object that wants to multicast before a message is multicast and unlocked only after the message is sent to all the members.

In the synchronous form of multicast, multiple forms of deadlocks are possible- eg., an object might wait on itself to form a deadlock or two objects might wait simultaneously for each other, forming a deadlock. We have used the following protocol to ensure deadlock free all source ordered multicasting. All user objects that want to multicast, follow this protocol.

$< Protocol >$

- Get the lock for the buffer to store the message in the group.
- Process any multicast jobs that are already sent to the user object concerned

- Send the message to all present group members
- unlock the previously locked buffer
- Now wait until the message is finished processing by all members for synchronous multicasting

The header file for this class along with the detailed descriptions of interface is given in Appendix A.4.

3.5 Event Manager/Handler

A basic event may be generated due to method invocations. However an event may generated by other objects or external agents like time events, mouse events etc. In case of external window events like mouse events WindowClass finds the visual object which should receive the event and directs the event accordingly.

First, we describe how the events are raised by the system for all basic events. VC++ generates the necessary code to raise the method execution events. It uses the *precond* specification of ECA rules to raise only those events which are of interest to some objects in the system. Time events are generated by two time handlers. One for single shot time events like *after 0:0:5* and *at 5:0:0* and the other for multiple shot time events like *every 0:0:5*. The single shot time handler maintains an ordered list of time events(increasing) required by all the objects. Rather than raising the system event at a constant rate, we set the timer at the earliest time in the ordered list.

The repetitive time handler generates a schedule for all such *every* time events and raises the appropriate time event to the objects that are interested in *every* time events from the prepared schedule. If any new *every* time event occurs the schedule has to be incrementally reconstructed. The algorithm to construct the schedule is explained by considering an example. If we want to raise the time events after *every* 0:0:2, *every* 0:0:3, and *every* 0:0:4, in that case we prepare a schedule for every 12 secs and the same schedule will be repeated, every 12 secs is the LCM of the three time periods. LCM of the periods gives the cycle length of a periodic schedule. The

possibility of next event in the schedule is GCD of the numbers. The granularity of the VC++ clock to generate periodic events is set to the GCD value. We prepare a schedule using the above technique. If any new event occurs then in that case we need to reconstruct the schedule. The algorithm for this is, assume that the new *every* time event occurred at time t1 and its period is *'p'*. First find the LCM of all the every events including the new one. If new LCM is greater than the old LCM then in that case repeat the previous schedule by k times, where k =new LCM/ old LCM. Will insert the new *every event* into the schedule from t1 in the schedule. If the new LCM is not greater than old LCM just insert the time event to the schedule. The new granularity value is equal to the GCD of old GCD , the new every event time-period 'p' and the offset in the schedule(i.e. the position of access in the schedule).

External events are handled by the WindowClass. Window class receives the external events that are raised by the mouse. It processes the mouse x, y positions and determines in which object bounding box the event happened, and it posts the appropriate event to that visual object.

Every object is associated with an event handler. The event handler for an object manages the events and execution of eca rules for that object. The event manager handles all the basic and composite events. Handling of basic events is straight forward, but to handle composite events we need to maintain the history of the events that are posted to a single object. We are considering an incremental model for detecting the composite events rather than maintaining an explicit event history. The composite events can alternatively be expressed as regular expressions [GJS], their occurrence can be detected using a finite automaton. The finite automata constructed for all the eca rules are used by the event manager to keep track of composite events and to process eca rules. For each eca specification, the automata will have a set of initial states, a set of current states, a set of final states, and a set of partial final states. The partial final states are associated with some actions, and those are used to process time composition operators and count repetitive operators. The action associated with the final states is the action that is to be performed if the condition of the ECA is satisfied.

In the following we construct finite automaton M_E for a given event expression E by induction on the structure of E.

We denote a finite automaton M_E by a 7-tuple(S,$\sum$,σ,I,F,P,$\prod$), where S is a set of states,

$\sum$ is a set of inputs(i.e. basic events),

σ : S x ($\sum \cup \{\lambda\}$) $\rightarrow$ S is a transition function,

I is a set of initial states,

F is a set of final states,

P is a set of partial final states, and

$\prod(p)$ is a set of actions associated with partial state(s) and final state(s).

- simple event a: M_E has three states: a start state s_o(non-accepting), p (accepting), and q(non-accepting). S = {s_o, p, q}. Figure 2 shows the transition diagram.

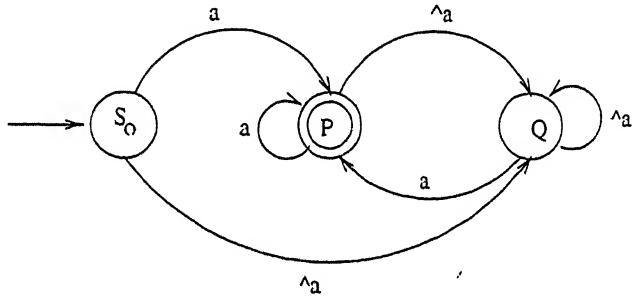


Figure 2: Transition diagram of event 'a'

- *any* event: M_E has two states: start-state and final-state. On any primitive event the move from the start-state is in to final-state. Figure 3 shows the transition diagram.

- E_1 && E_2: Let M_1 and M_2 be the finite automata of E_1 and E_2 respectively. M_E is constructed from M_1 and M_2 as follows. The start state of M_E is a pair rather than one state and also the final state is also a pair. M_E maintains a pair of current states. On every event both the current-states will move and the action is performed if both current-states reach the final states. The partial final states of M_1 and M_2 will be as it is to M.

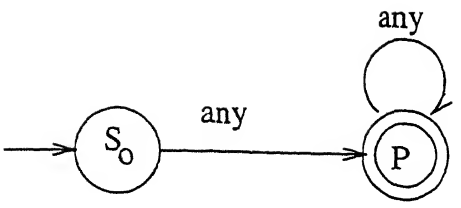


Figure 3: Transition diagram of 'any' event

Mathematically, S = S_1 x S_2, I = I_1 x I_2, F = F_1 x F_2, P =$P_1 \cup P_2$,
$\sigma((s_1, s_2), a) = (\sigma_1(s_1, a), (\sigma_2, a))$.

- $\sim$ E: Let M_1 be a deterministic automaton for E. The automaton for $\sim$E is obtained from M_1 b by switching the roles of accepting and non-accepting states, except for the start-state which remains non-accepting. The partial final states won't change.

  Mathematically, S = S_1, I = I_1, F = S - {I $\cup F_1$ }

- E_1 || E_2: This can alternatively be expressed as $\sim(\sim E_1$ && $\sim E_2)$. So the automaton methods of && and $\sim$ can be used to construct automata for disjunction operator.

- relative(E_1, E_2): Build two automata M_1 and M_2 for E_1 and E_2, respectively. Connect the accepting states of M_1 on empty event transitions to the start state of M_2. The accepting states are the accepting states of M_2 and the start state is that of M_1.

  Mathematically, S = $S_1 \uplus S_2$, I = I_1, F = F_2, P = $P_1 \uplus P_2$,
  $\sigma = \sigma_1 \cup \sigma_2 \cup \{(p.\lambda) \rightarrow q / \ p \in F_1, q \in F_2 \}$

  Figure 4 shows the transition diagram.

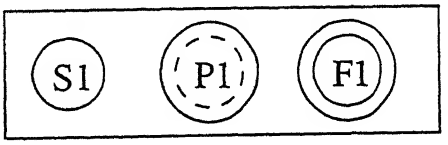


Figure 4(a): Transition diagram of E_1

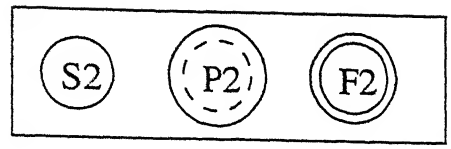


Figure 4(b): Transition diagram of E_2

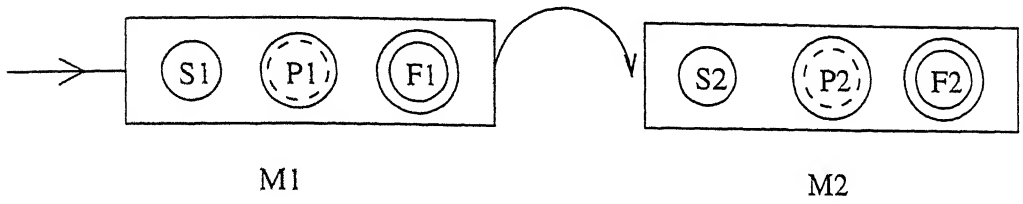


Figure 4(c): Transition diagram of relative(E_1, E_2)

- prior(E.F): This can alternatively be expressed as relative(E, any) & & F. The above automata construction methods of relative, any and conjunction operator can be used for prior.

- E_1 then time-period: Build an automaton M_1 for E. Let M be an automaton for "E *then* time-period". The final state of M_1 is treated as the partial final state of M. If M_1 has any partial final states that will be added to the set of partial final states of M. The action for the new partial final state is to request the time-handler to raise a time event after the time-period specified in the event. It creates two states namely wait and final. The final state of M_1 will go to the wait state on empty transition. The wait state will go to the newly created final state only when the requested time event happens. If any other event happens it will be in the wait state.

  Mathematically, S = $\{S_1 \uplus \{w, f\}\}$,
  $\sum = \sum_1 \cup \{$ timevent now+time_period 'e' $\}$
  P = $P_1 \cup F_1$,
  $\prod = \prod_1 \cup \{$ q $\mapsto$ raise time_event 'e' , q $\in F_1 \}$, F = {f},
  $\sigma = \sigma_1 \cup \{ (f_1, \lambda) \mapsto$ w / $f_1 \in F_1 \} \cup \{$(w, e) $\mapsto$ f $\}$ Figure 5 shows the transition diagram.

- relative(n,E_1): Build an automaton M_1 for E_1. Let M_E be the automaton of "relative(n, E_1)". The final state of M_1 is treated as the partial final state of

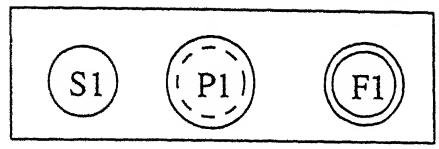


Figure 5(a): Transition diagram of E_1

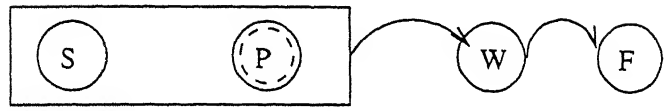


Figure 5(b): Transition diagram of E_1 then time_period

M_E. If M_1 has any partial final states that will be added to M_E. The action for new partial final state is, it increments the counter and checks whether it is greater than or equal to 'n' specified in the event. It creates a new final state. The automata will come to the newly created final state only when the above condition is satisfied. The new final state is connected to the start state(s) using empty transitions. Similarly it follows for the rest of the count repetitive operators.

- $+E_1$: Build an automaton M_1 for E_1. Let M_E be an automaton of "$+E_1$". The automata of "$+E_1$" is same as that of E_1 except that there is an extra arc from the final state(s) to start state(s) of M_1. Figure 6 shows the transition diagram.

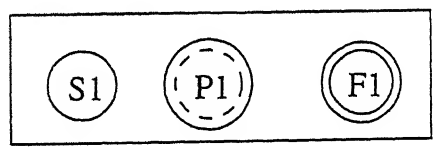


Figure 6(a): Transition diagram of E_1.

Events are monitored as follows : whenever a basic event is posted to an object, first it moves the current-position(s) of all the automata with the event posted, and checks the new current position(s) with the partial final states of the automata, if it matches, performs the actions that are associated with the partial final states and moves the current position and if it doesn't match with any partial final states then

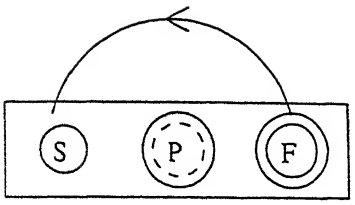


Figure 6(b): Transition diagram of $+E_1$.

it checks whether it is matching with any final-states. If it matches perform the associated action with that final-state(s) i.e. the C-A part of the ECA rule.

Chapter 4

VC++ Parser

VC++ is built on top of C++. It provides constructs that help develop visual simulation environments and domain specific visual programming languages. VC++ borrows and extends the object definition facility of C++ called class. So all valid C++ statements are also valid statements in VC++. The VC++ parser [ASU90], [VRG86] is built as a preprocessor that converts all valid VC++ specific constructs in a VC++ program into a semantically equivalent C++ code and plugs in the remaining C++ code intact. The parser is built using the compiler constructing tools *flex* and *bison*. The resulting C++ code can be compiled and linked with any standard C++ compiler. *GNU g++* compiler is used for compiling the resulting C++ code. A block diagram of the parser is shown in figure 11.

All the valid VC++ specific constructs the parser recognizes can be grouped into four types. They are

- Visualization Constructs

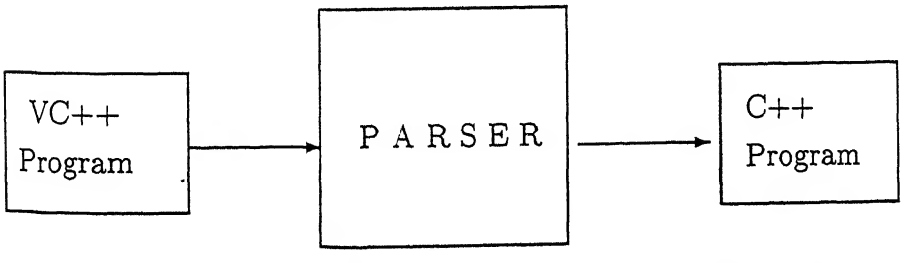


Figure 11: Block Diagram of the VC++ Parser

- Concurrency Constructs
- Communication Constructs
- Constructs for specifying reactive behaviour
- Implicit Constructs

For the sake of completeness all the constructs of VC++ that were implemented earlier by Singuru have also been included.

4.1 Visualization Constructs

These constructs are used to render visualizations for user objects. The resulting code represents the semantics of the program. There are four visualization constructs *param*, *components*, *views*, and *owndisplay*.

param: The *param* construct is used to declare a list of *display param variables*. It can be used for both variables inherited from the base class as well as variables newly created within the current class. The parser creates a temporary variable for each param variable and also records them in its data structures for subsequent handling events which are raised due to the changes in display param variable in the program code. This construct was implemented earlier by Singuru.[Kri96]

components: A complex user object may be represented using a number of basic component objects. The *components* construct can be used to indicate the basic components from which the main object is built. The parser inserts the necessary *inline method* code to create all the components and pushes them into the component list. This inline method will be called by routines like *move* which move the main object, by actually moving each individual component object and default *display* method to display the components. The list of components are accessible to the user. The user has the flexibility to operate on this list by using the methods that are specified for components.

views: A user's object may have a number of views/frames associated with it. The *views* construct is used to declare a list of views and the corresponding constructors with which they have to be created. It results in the parser generating a public *inline method*, that actually creates the views. The parser also inserts the necessary code to create an appropriate pointer type variable instance for each view indicated by the user. It maintains the current view as the first object in the list specified. It gives the user to operate on the list with the functions that were specified in chapter 2 for views.

owndisplay: This construct gives the flexibility to the user to have his/her own display function. If the user has not specified this construct before class declaration then the VC++ parser creates a default display function for that class. The display method for class containing components just iterates on the list of components by calling the display method associated with that component. In case of views it calls the display method for the current view.

4.2 Concurrency Constructs

These constructs are used for declaring and creating concurrent visual objects. The parser takes care of the necessary modifications to produce the required semantics of the program. We have the two concurrency constructs namely, *parallel_class* and *concurrent*.

parallel_class: The *parallel_class* construct is used to indicate that the class being declared is amenable for concurrency, i.e., object instances of this class can be declared as concurrent objects. The parser generates a stub routine for all methods other than constructors within this class. The stub routine do the *de-normalization* of the arguments in method invocations and call the corresponding original method with the proper arguments. This facility was implemented earlier by Singuru.

In our implementation we provided a new message exchange facility named async with timeout has been provided. In order support this another stub will get generated for each method with two more arguments namely timeout and the time at

which the event is raised. The VC++ parser inserts the necessary code to check whether timeout has elapsed or not. According to that the method will get invoked.

concurrent: All object instances of parallel classes, may not be parallel. A user can declare an object instance of a parallel class as concurrent, using the *concurrent* construct. This can be used for creating both simple as well as pointer type of concurrent objects. The parser does the necessary code insertions to create and allocate a unique *Mach thread*, for all such objects. This facility was implemented earlier.

4.3 Communication Constructs

Message passing is vital for inter object communication and these constructs provide the required facilities. VC++ provides the same *syntax* for communication constructs as it is for method invocations in C++ i.e., the syntax for sending a message to an object is same as the syntax for calling one of that object's method. This helps the programmer to use the concurrent objects in the same transparent way as sequential objects. The parser keeps track of all parallel object method invocations (i.e., message passings) and inserts the necessary code that schedules the message to the appropriate destination object. All this facilities are supported earlier by Singuru except async with timeout. VC++ parser adds two more arguments to the method async with timeout, i.e., timeout and the time at which the method is posted to the visual object in async mode.

4.4 Reactive Behaviour Constructs

The constructs are used for specifying the reactive behaviour pertains to an objects response an event of interest happens. The syntax for specifying the reactive behaviour is specified in Chapter 2.

The parser keeps track of all the method execution events that are interested and it inserts the appropriate code in methods to raise the events. The parser creates a

new class for each class that contains ECA rules. This newly created class variables maintains the event history i.e., current_states, initial_states, final_states and a flag to indicate if the event is happened. The class that contains the ECA rules inherits this newly created class.

The Parser collects all the composite events and generates the automaton for each composite specification and it also generates the separate event handler for each class. This event handler maintains all the ECA composite evaluations. The VC++ library provides the necessary routines for moving into the next state after an event of interest happens and also to move onto next states on empty transitions for the event handler.

The parser collects the required time events for each class and it initializes the both the single shot and multiple shot time handlers. The parser collects all the mouse button events of a particular class and creates a virtual function *is_of_interest.* This function takes one argument, i.e. mouse button events and it returns true if the mouse button event is of interest to that class. This function will be used by the WindowClass service management.

4.5 Implicit Constructs

In addition to the above *explicit* constructs, VC++ parser assumes certain constructs within the user's program as *implicit* constructs to achieve the desired VC++ provided semantics (i.e., to do some additional house keeping duties).

Whenever a new user's object is created within the program, its creation should be informed to the WindowClass. New objects can be created using the *new* constructor for pointer type of objects and by simply declaring the object, for non pointer type of objects. The parser inserts the appropriate statements that inform the object's presence to the WindowClass, after all such object creation statements within the program code.

Also the declaration *main()* which implies start of the main part of the user's program code is changed to the string *user_main()*, which will be provided as a callback routine to the start button in the user interface code.

VC++ parser creates a new function *detach_threads()* for terminating all the threads. This routine is provided as a callback routine to the done button in the user interface code. This actually posts the system defined detach_thread to the objects to terminate. This allows the objects to finish if there were any jobs left in the buffers.

VC++ parser creates a method *set_job()* for all classes if the classes are interested in either the mouse button events or time events. This method checks whether the event is mouse button event or time event, if it is mouse button event it again calls the *is_of_interest()* method to confirm whether the object is interested on this mouse event or not. If it is interested then it posts the event to the object. If it is not mouse event, if it is time event then in that case it justs posts the event to that object.

Chapter 5

Case Studies

VC++ is an adequate framework, on which a number of different visual environments can be developed. In this chapter, we considered two visual environments namely *Agentsheets*[RS95] and *LabView* as case studies.

Agentsheets is a Macintosh application implemented in Common Lisp. It is a system which supports the collaborative design of domain oriented visual programming languages in areas such as art, artificial life, education and environmental design. It provides role specific views and tools to meet the specific needs of designers. In section 3.1, we describe how the VC++ framework can be used to develop the tools supported by Agentsheets.

LabView is a visual programming application language. In LabView programs are called virtual instruments because their appearance and operation imitate actual instruments. A virtual instrument consists of a front panel and a block diagram for specifying its components in G, a data-flow based visual programming language. The front panel defines the user interface syntax, and the block diagram defines its semantics.[KASC95]

The LabView includes libraries of functions and development tools designed specially for instrument control. In the following section, we describe how VC++ framework helps in constructing the LabView. LabView provides a number of primitive box types/icons for specifying the program constructs. The primitive icons are nodes, terminals, and wires. Section 3.2 deals with a sketch about how VC++++

may be used to implement the functionalities provided by LabView.

5.1 Agentsheets on VC++

Visual languages created with Agentsheets consist of autonomous communicating *agents*. These agents are software routines that wait in the background and perform an action when an event occurs. Designers create domain oriented visual programming languages by defining the look and behaviour of agents.

There is a direct correspondence between the agents of Agentsheets and the visual communicating objects of VC++. In an object-oriented design, attaching a view corresponds to attaching a class. Designers declare and define the required *parallel classes*, of which the concurrent objects are direct instances. The objects of VC++ are more efficient than the agents of Agentsheets since in VC++ each concurrent object is alloted a unique thread which can run simultaneously as any other concurrent object. In contrast, each agent in Agentsheets is a separate process which involves lots of overhead.

In Agentsheets, agent communication can be explicit using links between agents or implicit, based on spatial relationships between the different agents. These features can be implemented using the synchronous and asynchronous message facilities provided by VC++ for object communication. VC++ provides the same syntax for communication, as it is for method invocation.

Agentsheets provides tools for creating components like cars, buses, roads etc. by assembling them from smaller components (i.e., for incrementally defining the look of the agents). These tools can be implemented using the *components* of VC++. Also the tools that provide for changing the various components dynamically like changing the topology of streets, can be implemented using the *push_component* and *remove_component* operations of VC++. Further certain attributes like car size, light frequency etc. can be declared as *param* variables of VC++, so that VC++ takes care of their implicit changes within the program.

Agentsheets also provides tools for querying particular components, and for providing many depictions and switching between these depictions for some components

to represent the underlying agent's state changes etc. These can be implemented using the *views* of VC++. The various depictions of a component can be declared as the views of the main view. VC++ provides operations that can be used to change the number of views (i.e. depictions) of a component, dynamically. Further it provides operations like *zoomin* and *zoomout* for switching between the various depictions.

AgentTalk editor of Agentsheets is Object Oriented and provides for incrementally defining the behavior of agents. This is directly supported by VC++, since VC++ is an extension of C++. Further Agentsheets provides features for declaring the behavior of agents directly, using simple AgentTalk code. For example, cars have to follow roads, watch out for other cars, avoid collisions, obey traffic signals etc. The behavioral declaration of the objects can be declared using ECA facility of VC++. ECA rules can be used for specification and analysis of reactive behaviour of objects.

Further, VC++ provides features for color and image allocation, move etc., that help in the rapid picture generation (visualization) and animation of the agents. The novel aspects of VC++ are it allows the user to specify his/her own visualizations, supports a user collaborative approach for designing domain oriented visual programming languages, and it makes easy development of a wide range of VPL/VPE.

5.2 LabView Using VC++

Consider an instrument built using LabView i.e. addition and subtraction.It is shown in figure 7. In figure 7; addition, subtraction, and databox refer to nodes; entry and exit ports refer to terminals. The lines that are used to connect the nodes and the terminals indicate wires in LabView. In VC++ we can implement each icon/box as a separate parallelclass for the nodes and the terminals are as the members of the class. The connection between the nodes and the functionality can be represented using ECA rules.

In the above example, after the data is read into the databoxes, it has to execute the functions *add* and *subtract* immediately. This can be achieved through the

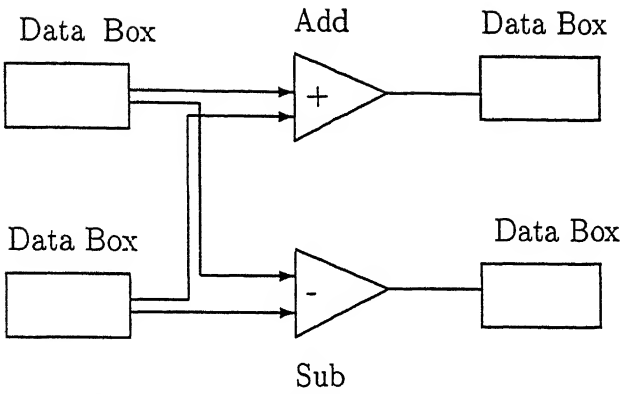


Figure 7: Instrument constructed using LabView

message facilities and ECA rules. In order to build the complete LabView we need to build classes for every individual node in LabView. The terminals can be included in the class definitions. The functionalities of connecting the terminal nodes to terminal nodes can be specified using ECA rules. The class declarations in VC++ for the above instrument is given in appendix-B.2.

Chapter 6

Conclusions

An important goal of research in programming languages is to provide a better fit between problem and program notation. We have tried in our work to do this for programming visual simulation environments and also for constructing domain specific visual programming languages. We developed a prototype of our language framework and which was tested by developing visual simulation environments namely Eco-system Pond and City Traffic Simulation. We also developed a gradesheet problem to explain how VC++ can be used to construct domain specific VPLs.

In this thesis we provided basic framework for handling events and extended the facilities to the classes that were defined earlier by Singuru. In the following we summarize the salient features of the VC++ system.

- It supports how to model a complex object using components.
- It supports multiple views to an object, by this user can have good animation by maintaining all the frames of the object.
- In addition to synchronous and asynchronous facilties it supports asynchronous with timeout.
- It supports external events i.e., mouse events and time events.
- It allows the non_member of the group to use sync multicast facility.

- One of the novel aspect of is allows the users to have his/her own display functions.

6.1 Future Scope

We considered adequate set of event composition operators. Further enhancements, both in terms of ease of use and in terms of expressive power, to event expressions can be made.

Similarly, in composition event evaluation, we are constructing non-optimized non-deterministic finite automaton, one can build an optimizer for the automaton generation.

By building a Graphic Language on top of it, one can convert VC++ into a pure visual language. By this we can increase the productivity of the user for the construction of visual programming languages.

Some of the listed extensions in Singuru's thesis not addressed in the current implementation also remain to be done.

Finally the parser can be generalized to provide more flexibility to the user. In the present implementation, we only providing location information for errors. Another extension can be to build a full-fledged compiler for VC++.

Bibliography

[ASU90] Alfred V. Aho, Ravi Sethi, and J. D. Ullman. *Principles, Techniques and Tools.* Addison-Wesley, 1990.

[BW95] Alan Burns and Andy Wellings. *Concurrency in Ada.* —, 1995.

[cP86] Man chi Pong. A graphical language for concurrent programming. *IEEE Computer Society Workshop on Visual Languages. IEEE CS Order No. 722. Dalls, Texas*, pages 26–33, June 1986.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, R. Ladin D. McCarthy, A. Rosenthal, and S. salin. The hipac project: Combining active databases and timing constraints. *ACM-SIGMOD Record 17*, pages 51–70, March 1988.

[Dec] *DEC OSF/1 Thread Programming Reference Manual.*

[ea86] T. D. Kimuara et al. A visual language for keyboardless programming. Technical Report TR WUCS.86.6, Dept. of Computer Science, Washington Univ. St. Louis, June 1986.

[ea88] D. Ingalls et al. Fabrik: A visual programming environement. In *Proc. ACM Conf. Object Oriented Programming, systems, Languages, and applications(OOPSLA 88), ACM, New York*, pages 176–190, 1988.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Welsey, 1990.

[GJS] N. H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation.

[GJS92] N. H. Gehani, H.V. Jagadish, and O. Shmueli. Event specification in acitve object-oriented database. In *Proc. ACM-SIGMOD 1992 Int'l Conf. on Management of Data, San Diego*, 1992.

[Hai59] Lois M. Haibt. A program to draw multi-level flow charts. In *Proceedings of the Western Joint Computer Conference, San Francisco, CA.*, volume 15, pages 131–137, March 1959.

[JR94] Johnson and Reichard. *X Window Applications Programming.* BPB, 2 edition, 1994.

[KASC95] T.Dan Kimura, Ajay A., S. Sengupta, and Julie W. Chan. Form/formula : A visual programming paradigm for user-definable user interfaces. *Computers, IEEE*, March 1995.

[kC86] Shi kuo Chang. Visual languages : A tutorial and survey. *Lecture Notes in CS, 282*, 1986.

[Kri96] Singuru R. Krishna. A visual object oriented programming environment. Master's thesis, I.I.T. Kanpur, 1996.

[MD89] D. R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM-SIGMOD 1989 Intl.Conf. Management of Data*, pages 215–224, May and June 1989.

[MP85] S. Matwin and T. Pietrzykowsh. Prograph: A preliminary report. *Computer Languages*, 10(2):91–126, 1985.

[NT85] Nadia and Daniel Thalmann. *Computer Animation-Theory and Practice.* Springer Verlag, Tokyo, 1985.

[Rei86] Steven P. Reiss. Visual languages and the garden system. *Lecture Notes in CS, 282*, 1986.

[RS95] Alexander Repenning and T. Sumner. Agentsheets: A medium for creating domain oriented visual languages. *Computers, IEEE*, March 1995.

[Shu85] Nan C. Shu. Formal : A forms-oriented visual directed application development. *IEEE Computer*, pages 38–49, August 1985.

[SMLM91] Michael L. Scott, Brian D. Marsh, Thomas J. Leblanc, and Evvangeles P. Markator. First-class user-level threads. *ACM*, 1991.

[VRG86] Robert V.Rubin, Steven P. Reiss, and Eric J. Golin. Compiler aspects of an environment for programming by demonstration. *Lecture Notes in CS 282*, May 1986.

Appendix A

Interface Details

This appendix gives the *main* part of the header files for each of the four basic classes, together with a detailed description of the interfaces they provide.

A.1 VisualObjClass

```
void *operator new(size_t, void *p);
class VisualObjClass
{
public:
// constructors for this class
  VisualObjClass();
  VisualObjClass(int x, int y, int l, int w);

  // destructor for this class
  ~VisualObjClass();
  bounding_box(int x, int y);

   virtual void  display(GC gc);
   virtual void  erase_display();
   virtual void  set_job(int event, int time);
   virtual void  paint(int x,int y);
 virtual void list_of_views();

  virtual  move(int deltax, int deltay);
  virtual int threshold(int deltax, int deltay);

  virtual void
```

```
  list_of_components();
  push_component(VisualObjClass *object);
  remove_component(VisualObjClass *object);
  void add_view(VisualObjClass *object);
  void delete_view(int id);
  void set_current_view(int id);
  int  get_current_view();
  int  no_of_views();
  void  FindColor( Colormap colormap,
          XColor *colr);
  void  FindColorByName(Colormap colormap,
              XColor *colr, char *cname);
  XImage *MakeImage(unsigned char *data,
          int width,
          int height);
};
```

< InterfaceDescriptions >

bounding_box(x, y) :

This method is used by the other class namely WindowClass, to check whether or not a given point (x,y) lies within the bounding box contained by the visual object.

display(GC) :

This method is a virtual function that must be redefined by the particular object to suit the way it is to be displayed on the screen. The argument to this function is a value of type GC (X windows specific), with which the display is to be done.

erase_display() :

This is another virtual function that is useful for erasing an object's image on the display.

paint(x,y):

This is another virtual function that must be defined by the user to draw the visual object.

set_job(event,type):

This is another virtual function it is used by the window class to post the external events namely time events and mouse events. The type specifies whether the event is mouse event or time event. If it is time event then it just posts the event to the appropriate object. If it is mouse event it checks, whether the object is interested on this event or not. If yes then posts that event to that object.

list_of_views() :

This is a virtual function, the redefined version of which is automatically generated by the VC++ parser in the user's visual class, in response to user's declaration of static views within his class.

list_of_components() :

This is another virtual function, the redefined version of which is automatically generated by the VC++ parser in the user's visual class, in response to user's declaration of static components within his class.

Components means those elemental parts with which the actual object is made up of. For eg, a car has as its components a booty, tyres, lights etc. i.e.,. a car is made up of all these parts.

add_view(object),
delete_view(id):

These methods provide the user with a facility to add or remove views to/from his list dynamically at run time.

get_current_view():

This method provide the user to get the current_view of the object.

set_current_view(id):

This method provides the user to change the current_view to the specified view.

no_of_views():

This method allows the user to find the number of views associated with the object.

push_component(object), remove_component(object):

These methods provide the user with a facility to add or remove components to/from his list dynamically at run time.

move(deltax, deltay) :

This is another virtual function that erases the user's object at the current position and displays it in the new position shifted by (deltax, deltay). It provides a primitive form of *move* facility to the user. User can redefine this function, if his application needs a more complicated form of move.

threshold() :

This is another virtual function that is used in the move function, and in the parser generated code for display param changes. It indicates whether the changes are sufficiently high or not for the objects to be erased and displayed.

The next three methods are helpful for the user, if he wants to use color displays.

FindColor(colormap, color) :

This method allocates a new color specified by the RGB values in XColor struct, in the given Colormap, if the colormap is free. If the colormap is not free, it returns the closest color from the already allocated colors.

FindColorByName(colormap, color, color_name) :

This method provides the same facilities like the previous method, except here the user can specify the color in the form of a string instead of RGB values.

MakeImage(data, width , height) :

It is useful for objects which are represented using bitmap files. It creates an image of required depth, for the programmer.

A.2 WindowClass

```
class Win_dow
{
 public :
 // constructor
 Win_dow();

 // destructor
 ~Win_dow();

  Place_in_window(VisualObjClass *g);
  remove_from_window(VisualObjClass *g);

  display_all();
  Quit();

 VisualObjClass *find(int x, int y);
 win_click(int x, int y);
};
```

< InterfaceDescriptions >

Place_in_window(object) :

The statements invoking this method are inserted by the VC++ parser, after every new visual object is created in the program. This method pushes the given object onto the top of the list, it maintains for all visual objects and also displays the object on the window.

remove_from_window(object) :

This method does the opposite of the previous method i.e., it removes the object from the list and erases it from the window. This method is actually called from the destructor for objects in the VisualObjClass.

display_all() :

This method is called by the callback routine that takes care of expose events. It redisplays all the objects that were originally on the window before the expose event occurred, by calling each object's corresponding display function.

Quit() :

This is a normal exit program that terminates the program by calling the exit(0) function.

find(x, y) :

It locates and returns the object whose bounding box contains the given point (x,y), among the list of objects this class maintains. This routine is used by the next two routines for locating an object.

win_click(x, y):

These methods locate the particular object whose bounding box contains the point (x,y) using *find* and posts the mouse event to the object if it is interested.

A.3 ConcObjClass

```
class ConcObjClass {
 public :
     //constructor
     ConcObjClass(VisualObjClass *obj_ptr);

     membership_count_incr();
```

```
    membership_count_dcr();

    worker_sync(VisualObjClass *obj_ptr);
    set_job(fn_ptr method_ptr);
    return_sync();
    async_return_sync(VisualObjClass *obj_ptr,
          int buffer);

  worker_job(VisualObjClass *obj_ptr);
  do_fast_job(VisualObjClass *obj_ptr);
  do_all_fast_jobs(VisualObjClass *obj_ptr);

  set_async_job(fn_ptr method_ptr,
  VisualObjClass *obj_ptr);
  form_arglist(void *pointer,
 int arg_size);
  form_async_arglist(void *pointer,
                      int arg_size);

 inform_master();
 inform_worker();
 inform_async_job();

 check_jobs(VisualObjClass *obj_ptr);
 static _finish_work_stub(VisualObjClass *obj_ptr,
   char *arglist);
};
```

< InterfaceDescriptions >

set_job(), form_arglist(), inform_worker() :

These three routines are used by the scheduler thread to synchronize and write the appropriate message with proper arguments in the destination object's *normal* buffer and inform about a new job to the destination thread.

set_async_job(), form_async_arglist(), inform_async_job() :

These three routines are used by the scheduler thread for a similar purpose as the previous routines but to write in the *urgent* buffer.

inform_master() :

When a sender finds no room to write in the destination's buffer, it simply sleeps until space is available. The destination thread awakens such threads whenever vacant space is available, using this routine.

worker_sync() :

This routine is used by the *worker()* routine. It first executes the jobs in the urgent buffer if any, and then executes a single job from the normal buffer. If there are no jobs, it simply sleeps until a job arrives.

return_sync() :

This routine is used to synchronize a job in the normal buffer.

do_all_fast_jobs() :

This routine when invoked, executes all the jobs that are currently in the urgent buffer and then returns.

async_return_sync() :

This routine is used, when a sender of an urgent message needs to wait until it is processed by the receiver. This is the logical view. But what physically happens is that the sender thread instead of waiting idly for the reply, continues with the remaining jobs if any in its urgent buffer. This is done so as to avoid *deadlocks*.

_finish_work_stub() :

This routine when executed, causes normal termination of the thread executing it. This routine is scheduled as a normal job, after scheduling of all operations on that thread are over i.e.,. just before the method definition ends in all methods. In this way, we need not use any special thread termination routines, to terminate a thread.

A.4 Group Communication Class

```
template <class T>
class Group : public T {
 public :
Group(void);
      int join_group(ConcObjClass *member);
      int leave_group(ConcObjClass *member);

      set_job(fn_ptr method_ptr);
      form_arglist(void *pointer, int arg_size);

      void multicast();
      void sync_multicast(T *obj_ptr);
};
```

< InterfaceDescriptions >

join_group(), leave_group() :

These routines, as the name indicates are used by a member to join and leave respectively from a group. The argument is the object's identifier, which is preprocessed by the parser to the appropriate ConcObjClass instance alloted for the object.

already_member(), group_full() :

These are private routines, used by previous routines to check whether the given member is already present in the group and whether the group has place for new members to join or not.

set_job(), form_arglist() :

These routines are used to write the multicast message in the proper format in the group's buffer. The statements invoking these routines are inserted by the VC++ parser.

multicast() :

This routine provides a simple form of multicast. It provides an all source ordered form of multicast.

sync_multicast() :

This routine provides a synchronous form of multicast i.e.,. it doesn't return until and unless the message is processed by all the present group members. It ensures deadlock free multicasting.

Appendix B

Class Descriptions of Examples

B.1 City Traffic Simulation

To illustrate the use of VC++, here we have given the code for City Traffic Simulation. The problem of City Traffic Simulation is as follows: objects representing cars move forward when a stretch of road is in front of them and obey the traffic rules. The other objects in City Traffic Simulation viz. traffic lights keep on changing their states(red, green, amber) at a predefined rate. The detailed behaviour of the objects are given below.

The behaviour of the vehicle is :

- After the vehicle is started check whether it is about to collide with any other vehicle and also check whether the traffic light is green or not if it is closer to the traffic light.
- If the vehicle is about to collide with any other vehicle then don't move the vehicle.
- If the relevant traffic light is red then join the vehicle in the corresponding waiting group.
- If the vehicle is not going not to collide with any other vehicle and the relevant light is green then move the vehicle.
- If the vehicle reaches a four road junction take a random decision to go either in left, right or straight directions.

The detailed definition of the vehicle class is

```
owndisplay parallel_class Vehicle : public VisualObjClass
{
private:
        int light_on;
        int     collision;
        Light   *light1;
        Light   *light2;
        int     direction;
        int     turn;
        int     nearer_to;
        int     len;
        int     wid;
int car_id;
        object_data    obj_info;// object_information
public:
        Vehicle();
        Vehicle(int x, int y, int len, int wid,
               Light *l1, Light *l2, int direction, int near,int carid);
  // method displays the id of the car
void  Identify();
        // method to start the car
        void     start();
        // method to move the car
        void     move();
         // method for deciding the direction
        void     decide_direction();
         // method for checking light whether it is on or off
         // in the going direction
        void      check_light();
        // method to detect whether there is any collision
        void      check_collide(Vehicle *which_car, object_data finfo);
```

```
        // method to join waiting_group
        void        join_waiting_group(int direction);
        void      display(GC x);
ecarules:
        e_chk_lig, e_col_lig1, e_col_lig2, e_move,
        e_start1, e_start2, e_decide,button;
};
eca Vehicle :: button
{
precond:
post_to: self;
cond: Mouse_Button_1_down
action: sync Identify();
}
eca Vehicle :: e_start1
{
precond: after start
post_to: car_grp;
cond:
action: sync check_collide(self, obj_info);
}
eca  Vehicle :: e_start2
{
precond: after start
post_to: self;
cond:
action: sync check_light();
}
eca  Vehicle :: e_col_lig1
{
precond: after check_collide
        after check_light
post_to: self;
```

```
cond: (after check_collide, after check_light) &&
        (collision == FALSE) && (light_on == FALSE)
action: sync move();
}
eca Vehicle :: e_col_lig2
{
precond: after check_collide
         after check_light
post_to: self;
cond: (after check_collide, after check_light) &&
        (collision == TRUE) && (light_on == FALSE)
action: async start();
}
eca Vehicle :: e_move
{
precond: after move
post_to: self;
cond:
action: async start();
}
eca Vehicle :: e_chk_lig
{
precond: after check_light
post_to: self;
cond: light_on == TRUE
action: sync join_waiting_group(direction);
}
eca Vehicle :: e_decide
{
precond: after move
post_to: self;
cond: turn == CROSS
action: sync decide_direction();
```

```
}
```

The behaviour of the Traffic Light is :

- Keep on changing the state of the Traffic light at some predefined rate.
- As the light turns green start all vehicles which are waiting for the light to turn green.

The detailed definition of the traffic light class is

```
struct position
{
        int x,y;
};
owndisplay parallel_class Light : public VisualObjClass
{
private:
        int  lights[4];
        position  p_lights[4];
        int current;
        Group<Vehicle>*south_wait;
        Group<Vehicle>*north_wait;
        Group<Vehicle>*west_wait;
        Group<Vehicle>*east_wait;
        GC gc_r;
        GC gc_g;
        GC gc_a;
        int num;
public:
        Light(){}
        Light(int x, int y, int len, int wid,
              int sx,int sy,
              int nx,int ny,
```

```
                int ex,int ey,
                int wx, int wy, int num);
        friend class Vehicle;
        // Lights functioning starts
        void lights_start();
        // change lights
        void change_lights();
        void display(GC x);
ecarules:
        s_event, n_event, e_event, w_event, a_change;
};
eca Light :: a_change
{
precond: after change_lights
post_to: self
cond:
action: sync change_lights();
}
eca Light :: s_event
{
precond: after change_lights
post_to: south_wait;
cond: current == SOUTH
action: async start();
}
eca Light :: n_event
{
precond: after change_lights
post_to: north_wait;
cond: current == NORTH
action: async start();
}
eca Light :: e_event
```

```
{
precond: after change_lights
post_to: east_wait;
cond: current == EAST
action: async start();
}
eca Light :: w_event
{
precond: after change_lights
post_to: west_wait;
cond: current == WEST
action: async start();
}
```

In our VC++ the main() part looks as follows

```
Group <Vehicle> *car_grp;

main()
{
        road Road;
        car_grp = new Group <Vehicle>;
        l1 = new  concurrent
              Light(1,1,1,1,180,180,340,180,340,340,180,340,1);

        l2 = new  concurrent
              Light(1,1,1,1,180,180,340,180,340,340,180,340,1);
        car1 = new concurrent Vehicle(20,220,50,20,l1,l2,EAST,1);

        car2 = new concurrent Vehicle(220,20,50,20,l1,l2,EAST,2);

        car_grp->join_group(car1);
        car_grp->join_group(car2);

        l1->lights_start();
```

```
        l2->lights_start();

        car1->start();
        car2->start();

}
```

The VC++ preprocessor converts this main() into a call-back routine named user_main() of the start button of User interface provide by the VC++. It also creates one more call-back routine to stop button i.e. detach_threads(), this detaches the all threads and it allows all the threads to complete the pending method executions.When the program is run a simple user interface layout will be provided by the VC++ with two buttons Start and Stop. Once the user clicks on the start button, control is transferred to the user's main and the desired animation is seen on the window. The stop button is used to terminate the program.

B.2 LabView Instrument Class Description

The behaviour of the memory_box or data_box is as follows.

- Whenever it receives data from the keyboard it will post the data which is read from the keyboard and its id.
- Whenever it receives data from the operation_box it displays the data on the display.

The detailed class declaration of the memory_box is given below.

```
owndisplay parallel_class memory_box : public VisualObjClass
{
private:
      int             value;
      char            val_str[10];// for display purpose
      int             position; // specifies the id
      GC              temp_gc; // Graphic context of 'X' Windows
```

```
        int             length,width;
        Widget          text_t;
public:
        // constructors
          memory_box();
          memory_box(int pos,operation_box *p,
                     int x, int y, int l, int w,
                     int active);
        // read_data
          void    read_data(int x);
        // set_data
          void    set_data(int x);
        // display_data on the stdout
          void    display(GC x);
ecarules:
          m_event, m_event1;
};

eca memory_box :: m_event
{
precond: after  read_data
post_to: ope;
cond:
action: async  record_value(position,value);
}
eca memory_box :: m_event1
{
precond: after  set_data
post_to: self
cond:
action: sync  display(temp_gc);
}
```

The behaviour of the operation_box is as follows.

- Whenever it receives message from data_box it raises an user defined event. The user defined event raised by the operation_box is varied according to the message it received from the data_box.
- If it receives two different user defined events in any order, it performs the appropriate operation.
- Whenever it performs an operation, it sents the result to the result_box.

```
owndisplay parallel_class operation_box : public VisualObjClass
{
private:
        int             value1;
        int             value2;
        int             type; // operation type either +,-,/,*
        int             result;
        GC              temp_gc;
        memory_box      *mem;
        int             length,width;
        Widget          text_t;
public:
        // constructors
         operation_box();
         operation_box(int p,memory_box *q,
                       int x,int y,int l, int w);
         int      get_result();
         void     operation();
         void     record_value(int p,int value);
         void     display(GC x);
 ecarules:
         p_event, c_event;
 };

 eca  operation_box ::p_event
```

```
        precond: after  operation
        post_to: mem
        cond:
        action: sync set_data(result);
}
event E1;// E1 is a user defined event
         // it will be posted to the operation_box
         // object by the record_value method
event E2;
eca operation_box :: c_event
{
        precond: after record_value
        post_to: self
        cond:
                (E1,E2) || (E2,E1)
        action: sync operation();
}
```

CSE-1997-M-RED-VIS